View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Repository @ Iowa State University

IOWA STATE UNIVERSITY
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

1-1-2003

An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations

Sampath Dechu
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation

Dechu, Sampath, "An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations" (2003). *Retrospective Theses and Dissertations*. 19939.
https://lib.dr.iastate.edu/rtd/19939

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations

by

Sampath Dechu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Chris Chong-Nuen Chu, Major Professor
Akhilesh Tyagi
Soma Chaudhuri

Iowa State University

Ames, Iowa

2003

Copyright © Sampath Dechu, 2003. All rights reserved.

Graduate College
Iowa State University

This is to certify that the master's thesis of

Sampath Dechu

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this thesis, we present a fast algorithm to construct a performance driven routing tree with simultaneous buffer insertion and wire sizing in the presence of wire and buffer obstacles. Recently several algorithms[13, 10, 9, 16] have been published addressing the routing tree construction problem. But all these algorithms are slow and not scalable. here we present an algorithm which is fast and scalable with problem size. The main idea of algorithm is to specify some important high-level features of the whole routing tree so that it can be broken down into several components. We apply stochastic search to find the best specification. Since we need very few high-level features to evaluate a routing tree, the size of stochastic search space is small which can be searched in very less time. The solutions for the components are either pre-generated and stored in lookup tables, or generated by extremely fast algorithms whenever needed. Since, the solutions of the components can be constructed efficiently, we can construct and evaluate the whole routing tree efficiently for each specification. Experimental results show that, for trees of moderate size, our algorithm is at least several hundred times faster than the recently proposed algorithms[9, 16], with not much difference in delay and resource consumption.

CHAPTER 1. INTRODUCTION

As VLSI technology enters deep submicron era, interconnect delay becomes a dominant factor in determining circuit performance. Interconnect optimization techniques like buffer insertion and wire sizing have been shown to be effective in reducing interconnect delay [1]. In modern VLSI design, it is very common to consider buffer insertion and wiring sizing during performance-driven routing. For the routing of two-terminal nets and when there is no restriction on buffer positions in the routing area, the route with optimal delay can be constructed by inserting buffers and sizing wires of the shortest path from source to sink. In other words, routing and interconnect optimization can be performed sequentially. However, for multi-terminal nets or when there are macro blocks where wires can be passed but buffers cannot be placed, the optimal routing tree can only be found if routing, buffer insertion and wire sizing are considered simultaneously. Hence while doing performance driven routing, there are number of degrees of freedom which may be exploited including, *topology construction*, *buffer insertion*, *wiresizing*, *topology embedding*. The past ten years has been growth of a fairly substantial body of work in this area.

Many algorithms have been proposed in the past few years to construct routing trees

with buffer insertion and wire sizing in the presence of routing and buffer obstacles. The approaches used can be classified as either sequential or simultaneous approaches. In sequential approach, the routing tree is constructed (*topology construction and embedding*) followed by buffer insertion and wire sizing. In simultaneous approach, routing tree is constructed by simultaneously considering routing, buffer insertion and wire sizing.

The algorithm proposed in [3] follows the sequential approach. But it does not consider wire obstacles, buffer obstacles and wire sizing. In [8], as part of sequential approach, Hu *et al.* extended van Ginneken's algorithm to solve the problem of buffer insertion on a given routing tree, considering only buffer blockages.

Several algorithms have been proposed based on the simultaneous approach. Topology search based algorithms given in [13, 10, 9] limit the routing topology space to certain topologies and search exhaustively for the best solution in that limited space. The final routing tree obtained from these algorithms depends on the criteria used to limit the topology space and the initial routing topology given to these algorithms. In order to obtain a better solution, a larger topology space needs to be considered and the exhaustive search usually takes a significant amount of time. All these algorithms are not scalable and they cannot handle wiresizing.

For two-terminal nets, Zhou *et al.* [17] presented a dynamic programming algorithm for simultaneous routing with buffer insertion, considering both buffer and wire obstacles. Lai and Wong [11] formulated the simultaneous routing with buffer insertion and wire sizing in the presence of buffer and wire obstacles as a graph-theoretic shortest path

problem. However, these two algorithms cannot be easily extended to handle multi-terminal nets.

Recently, Tang *et al.* [16] presented an algorithm graph-RTBW for multi-terminal nets that considers buffer insertion, wire sizing, and buffer and wire obstacles simultaneously. In their approach, the routing problem is converted into a collection of graph problems. One graph is constructed for each subset of sinks. In the graph, each vertex represents a subset, and other vertices represent possible buffer choices at different buffer locations. The shortest path from the subset vertex to every other vertex v in a graph corresponds to the optimal subtree with appropriate buffer insertion and wire sizing connecting v and the subset of sinks. Dynamic programming is used to construct routing solutions for larger subset of sinks based on solutions for smaller subsets of sinks. Finally, the routing solution for all sinks is obtained. They use Rubinstein delay model[15] for interconnect delay calculation. As they consider all the subsets of sinks, the runtime is exponential to the number of sinks. Hence the algorithm is very slow.

In this thesis, we present a very fast and scalable algorithm named Fast-Route for solving this problem. The main idea of our approach is to specify some important high-level features of the whole routing tree so that it can be broken down into several components. We apply stochastic search to find the best specification. Since we need very few high-level features, the size of stochastic search space is small.As size of the solution space is small, the time required to search for high level specifications of the routing tree is very less. The solutions for the components are either pre-generated and

stored in lookup tables, or generated by extremely fast algorithms whenever needed. Since, the solutions of the components can be constructed efficiently, we can construct and evaluate the whole routing tree efficiently for each specification. Experimental results show that, for trees of moderate size, our algorithm is at least several hundred times faster than the recently proposed algorithms[9, 16], with not much difference in delay and resource consumption.

Our approach has the following advantages over previous approaches:

1. Fast-Route is much faster and scalable. We apply stochastic search on a *small* search space. The evaluation of a specific routing tree is also fast, because lookup tables and fast algorithms are used to find component solutions. Runtime of our algorithm decreases as number of blockages in the design increases, because we have to search less number of positions for buffers. The graph-RTBW algorithm uses exhaustive search by trying all combinations of subset of sinks, buffer positions and buffer sizes. As a result, the algorithm is very slow. Topology search based approaches try to reduce the runtime by limiting the search space to certain topologies. But these algorithms cannot handle wire sizing and are not scalable with problem size. These algorithms become slower as the number of blockages increases in the design.

2. Fast-Route does not have restriction on the fanout of buffers. The graph-RTBW algorithm theoretically can handle general fanout. However, the expensive term in the time complexity of the algorithm is $O((t+1)^k B^{t+1} N^{t+1})$, where t is the

bound on fanout, k is the number of sinks, B is number of buffer types, N is number of possible buffer locations. So, in practice, the algorithm can only handle a fanout of 2. The example given in figure 1.1 demonstrates the disadvantage of having a restriction on fanout. The figure 1.1 (a) shows routing tree obtained by graph-RTBW for $t = 2$ and figure 1.1(b) shows routing tree obtained by our algorithm. The delay of routing tree shown in figure 1(b) is 37.7% better than the delay of routing tree shown in figure 1(a). Also, we observe that routing resources are wasted in the first case.

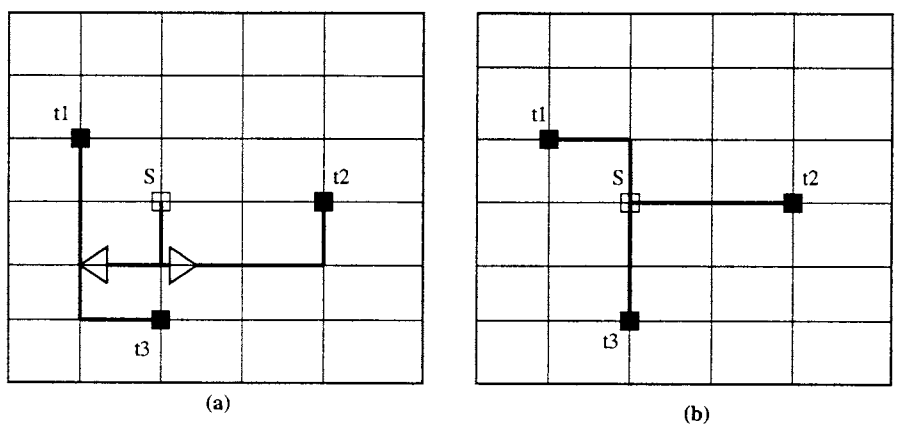

Figure 1.1 (a) Routing solution by graph-RTBW. (b) Routing solution by our algorithm

3. We use Elmore delay model[6] for calculating the delay of interconnects. It gives better estimation of delay when compared to Rubinstein delay model used in [16]. By using Elmore delay model, we can minimize the delay of critical path of multi-terminal net, which is not feasible by using Rubinstein model.

CHAPTER 2. Delay Model and Problem Formulation

2.1 Delay Models

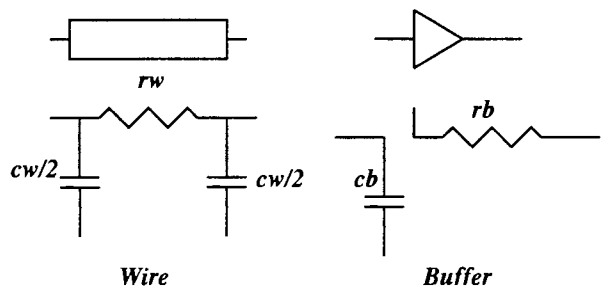

Figure 2.1 Delay Models

Elmore delay model has been a popular delay model since it was proposed in [6] because of its simple analytical form, fidelity and other properties [7]. We model wire segment as π type RC element, the Elmore delay of the individual wire segment is:

$$t_{wire} = r_w(\frac{c_w}{2} + c_d)$$

where c_d is the downstream capacitance for this wire segment. Similarly, for an inserted buffer, with input capacitance c_b, output resistance r_b, and intrinsic delay d_b, the Elmore delay is computed as:

$$t_{buffer} = r_b c_d + d_b$$

The total Elmore delay along any interconnect path is computed by summing together the individual wire delays and buffer delays. The figure 2.1 shows the the delay models.

2.2 Problem Formulation

The goal of the algorithm is to construct a routing tree with buffer insertion and wire sizing in the presence of wire and buffer obstacles, such that the maximum delay from source to sinks is minimized. Without buffer obstacles, the optimal routing tree is a shortest path tree rooted at the source node. But in the presence of macroblocks, which are available for wiring but infeasible for buffer insertion, the shorest path tree doesnot guarantee to be the optimal routing tree with minimized maximum delay from source to sinks. Previous algorithms which ignore macro blocks during routing tree construction may result in a very inferior solution. We illustrate the importance of the problem by an example (given in [16]) by an example of the 3-pin net in Fig 2.2. Shaded boxes represent routing obstacle regions where wiring is no allowed, and gray boxes represent macro blocks where buffering is not allowed. The routing grid has unit length of $0.5mm$. The technology parameters are taken from [19] with unit wire resistance $0.76\Omega/\mu m$, unit length capacitance $0.118fF/\mu m$, driver resistance of source and output resistance of buffer 180Ω, load capacitance of sink and input capacitance of buffer $23.4fF$, and intrinsic delay for buffer $36.4ps$. An optimal shortest path tree is shown in figure 2.2(b), and its delay is $1183ps$ with proper buffer insertion. As a result, it produces large delay. An alternative approach is to regard macro blocks as routing obstacles and avoid wiring

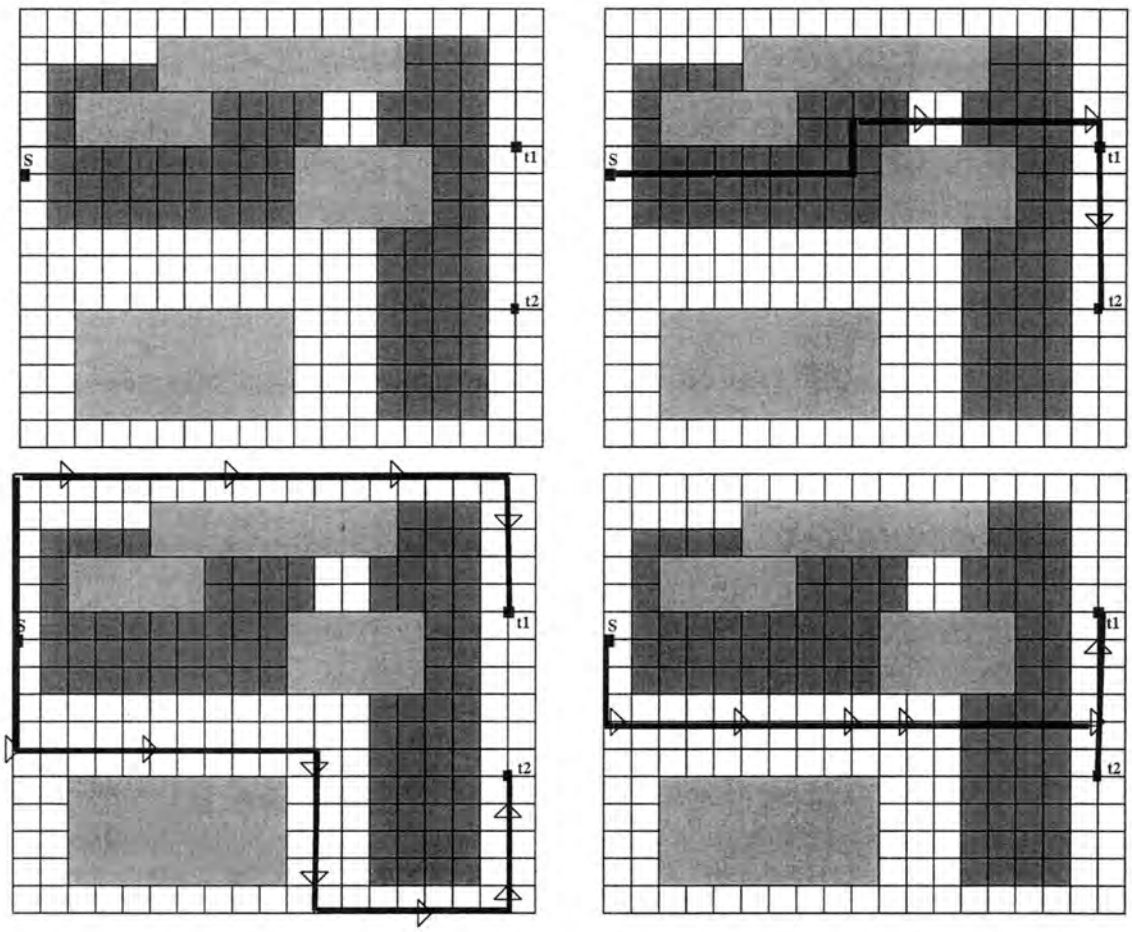

Figure 2.2 (a)Routing tree problem (b) Shortest path tree + buffer insertion, delay = 1183ps (c) Avoid obstacles+buffer insertion, delay = 1220ps (d)Simultaneous Approach, delay = 977ps

over them. An optimal solution by this approach is shown in figure 2.2(c), and its delay is 1220ps. Although buffer can be inserted anywhere, it wastes routing resource and may be too long to have short delay. By wisely routing with proper buffer insertions, we can obtain a better solution which is shown in figure 2.2(d) with delay 977ps.

In the problem of routing tree construction, the routing area is represented by a grid graph $G = (V, E)$. Routing obstacles and buffer obstacles are present in the routing area. Each edge E is a wire segment. Wire library W provides wire choices, and buffer library B contains buffer choices to be inserted at grid nodes in G.

Problem: *Given a routing grid graph* $G = (V, E)$ *, a buffer library* B*, a wire library* W*, a source node* $s \in V$ *and* k *sink nodes* $t_1, t_2, \ldots, t_k \in V$ *of net, find a buffered routing tree* T *rooted at* s *and leafed at* $t_i, i = 1, \ldots, k$ *, for each node* $v \in T$, $b(v) \in B \bigcup \{0\}$ *where* $b(v) = 0$ *indicates no buffer is inserted at* v *and* $b(v) \neq 0$ *requires* v *to be a buffer node, for each segment* $l \in T$ *wire* $w(l) \in W$ *, such that the maximum delay from* s *to sink* t_i *is minimized.*

CHAPTER 3. THE FAST-ROUTE ALGORITHM

In this section we define some of the terms we use and explain Fast-Route algorithm in detail.

3.1 Notation

We use same notation given in [16]for the following terms:

- W : Library of different wire types.
- B : Library of different buffer types.
- V : All nodes present in grid graph G.
- $N = \{s\} \bigcup \{t_i \,|i = 1, \ldots, k\} \bigcup \{buffer\, nodes\}$: Set containing source, sink and buffer nodes. Buffer nodes are the nodes where buffers can be placed. Clearly, in the presence of obstacles, number of elements in N is less than number of elements in V , which is set of all nodes in grid graph.
- **Wire Path:** A path connecting two nodes in N by properly sized wires but no buffers between.

- **Buffered Path:** A path connecting two nodes in N with buffers inserted between them. A wire path is a special case of buffered path.

We introduce the following new terms :

- **Leaf Buffer Path:** A buffered path connecting a buffer node directly to a sink without any wire branch in between. Leaf buffer path can be of zero length. A sink will be a leaf buffer path with zero length, if there is no buffer driving it directly without any wire branch in between.

- **Branch:** Branch is a tree component connecting three or more nodes in N, without any internal buffers. Every branch contains a driver buffer which is driving the branch and several receiver buffers which are connected to driver. Number of receivers in branch is called degree of branch denoted by t . In [16], branch is referred as Buffer Combination.

- **Stem Buffered Branch:** A branch which can have buffers on stem. Every branch can be considered as stem buffered branch with no buffers on stem.

- **Component Driver**: Buffer driving a stem buffered branch or a leaf buffer path is called component driver.

Figure 3.1 shows, the notations that we use in this paper. In the figure, the source is called the component driver, because we assume that the driver resistance of the source equals one of the resistance of buffers in buffer library B and the load capacitance of the

sinks equals one of the capcitance of buffers in B . If not, we can always add additional buffer types with source resistance and sink capacitance in B and letting the buffer type to be used only at source and sinks.

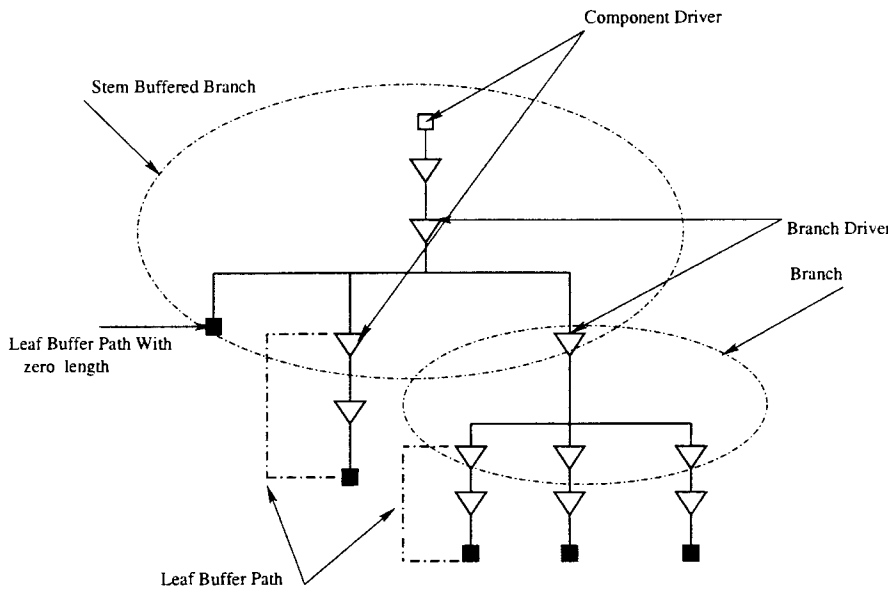

Figure 3.1 Illustration of Notations

3.2 Decomposition of Routing Tree

From the figure, we can observe that any routing tree is a set of branches and connecting buffered paths. A buffered path is a set of connected wire paths. Both branches and buffered paths can be precomputed. Given a routing grid graph $G = (V, E)$, let $N = \{s\} \bigcup \{t_i | i = 1, \ldots, k\} \bigcup F$, where F is the set of nodes where buffers can be inserted. Clearly, $|N| \leq |V|$. For a grid graph G with a lot of buffer obstacles, $|N|$ is much less than $|V|$. For any $u, v \in N$, the minimum delay of a wire path from u to v is a simple function of the shortest distance from u to v(denoted as $d(u, v)$) in G,

the driver resistance in u, the load capacitance in v and wire library W. Since the buffer library B is given, the possible driver resistance and load capacitance are known. Buffer to buffer wire sizing solutions can be precomputed and stored in a table for wirepath. The minimum delay values from u with buffer b_u to v with buffer b_v can be found by looking up the wire path table. It should be noted that we assume the driver resistance of the source equals one of the resistances of buffers in B and the load capacitance of the sinks equals one of the capacitance of buffers. If not it can be easily solved by adding an additional buffer type with source resistance and sink capacitance into B and letting the buffer type to be used only at source and sinks. After getting wire path table, we can compute the optimal buffered path between any two nodes. We use dynamic programming approach with pruning to construct the buffered path table from wire path table. The delay of branch $(v, r_1, r_2, \ldots, r_t)$, where v is driver and $r_1, \ldots, r_t$ are receivers, is more complicated to compute. However, it is still a function of the distance configuration of the buffers in branch, the driver resistance of v, the load capacitance of $r_i, i = 1, \ldots, t$, and wire library W. As an example figure 3.2 shows a branch of three nodes (v, r_1, r_2). Its minimum delay is determined by the stem length e, branch lengths l_1, l_2, the driver resistance of v, the load capacitances of r_1, r_2, and the wire library W. In practice, the degree of branch$(t + 1)$ is small. Hence using some simple steiner tree algorithms we can easily precompute the branch solutions and tabulate them.

We precomputed and tabulated the delay of buffered paths between any two buffer nodes in V and braches connecting three or more buffer nodes in V. If we can find the

locations and sizes of the buffers in the routing tree, we can easily construct the routing tree using these precomputed look up tables. But as number of buffers in routing tree is sometimes large, finding locations and sizes of buffers becomes very expensive. To overcome this problem, we decompose the routing tree into

1. Stem buffered branch

2. Leaf buffer path

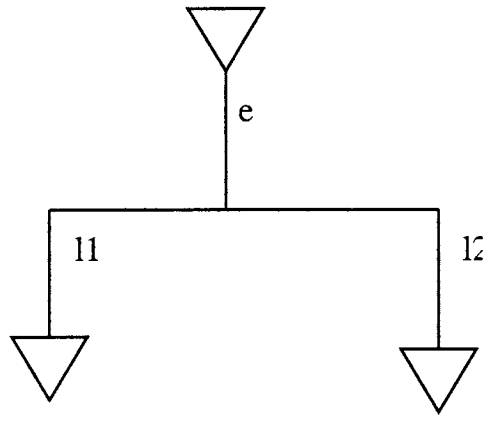

Figure 3.2 Branch

The delay of a leaf buffer path can be found using buffered path table. Since Stem buffered branch is a branch connected to a buffered path, its delay can be calculated using branch table and buffered path table. By making this type of decomposition we can reduce the number of buffers whose locations and sizes we need to find can be reduced. We call Stem buffered branch and Leaf buffer path as components of routing tree. If we can find the locations and sizes of the component drivers we can easily find the optimal routing tree using lookup tables.

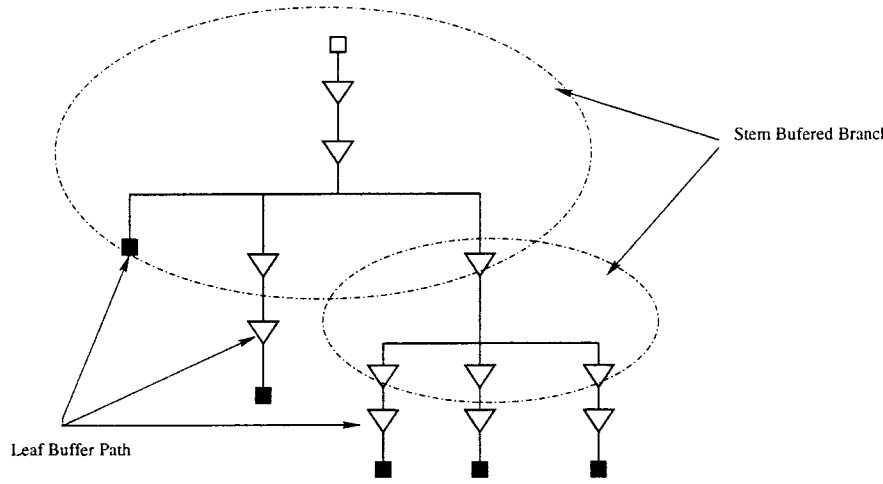

Figure 3.3 Routing Tree Decomposition

3.3 Component Construction

In this section, we explain how the solutions of components are pre-computed. As explained in the previous section, every routing tree consists of only stem buffered branch and leaf buffer path. Leaf buffer path is nothing but buffered path between a node in N and a sink. Stem buffered branch consists of two parts, one is branch and the other is buffered path between component driver and branch driver. Hence if we have pre-computed delays of buffered paths and branches, we can calculate the delays of components. We construct the delay look up tables as explained below,

3.3.1 Buffered Path Delay Table (BPDT)

Buffered Path Delay Table contains the delay of optimal buffered path between two nodes in N. If we know the sizes of buffers at buffer nodes and locations of those buffers, we can find the delay of the optimal buffered path between those two buffer nodes by

looking $BPDT(b_1, b_2, l_1, l_2)$, where b_1 and b_2 are buffer sizes, l_1 and l_2 are buffer locations.

To construct buffered path delay table, we construct the following tables:

1. Shortest Wire Path Length Table($SWPLT$) : This contains the shortest wire path length between two nodes in routing grid. We can represent this table in a functional form as $SWPLT(l_1, l_2)$, where l_1 and l_2 are the locations of two nodes. We use Lee's Algorithm to construct $SWPLT$.

2. Shortest Buffered Path Length Table ($SBPLT$): This contains the shortest buffered path length between two nodes in routing grid. We can represent this table in a functional form as $SBPLT(l_1, l_2)$, where l_1 and l_2 are the locations of two nodes. We use Lee's Algorithm to construct this table.

3. Wire Path Delay Table($WPDT$): This table contains the optimal wire path delay between two nodes in N, after wiresizing. We can represent this table in functional form as, $WPDT(b_1, b_2, L)$, where b_1 and b_2 are buffer sizes placed at those nodes and L is the length of the wire path length between these two nodes. We can get L from $SDTW(l_1, l_2)$. We use dynamic programming technique to construct $WPDT$.

4. Buffered Wire Segment Delay Table ($BWSDT$): This table contains the optimal delay of a wire segment, after placing the buffers on it. Assuming a buffer of size b_1 is driving a buffer of size b_2 through a wire segment of length L, Buffered Wire

Segment Delay Table can be represented in functional form as $BWSDT(b_1, b_2, L)$. We use dynamic programming approach to construct this table.

Using above tables and pruning technique given below, we construct $BPDT$ in a very short time. The algorithm below is to insert only a single buffer on a wire path. Step 3 should be repeated to insert more number of buffers.

step1: $BPDT(b_u, b_v, l_u, l_v) = BPDT(b_u, b_v, l_u, l_v)\ \forall u, v \in N$

step2: *//for inserting one buffer*

`for each` $w \in N$

`if(`$SWPLT(l_u, l_w) + SWPLT(l_w, l_v) < SBPLT(l_u, l_v)$ `)`

$BPDT(b_u, b_v, l_u, l_v)$

$= min(WPDT(b_u, b_v, SDTW(l_u, l_v)), BPDT(b_u, b_v, l_u, l_v))$

$SBPLT(l_u, l_v) = SWPLT(l_u, l_w) + SWPLT(l_w, l_v)$

step3: *//for inserting 3 buffers*

for each $w \in N$

`if(`$SBPLT(l_u, l_w) + SBPLT(l_w, l_v) < SBPLT(l_u, l_v)$ `)`

$BPDT(b_u, b_v, l_u, l_v)$

$= min(WPDT(b_u, b_v, SDTW(l_u, l_v)), BPDT(b_u, b_v, l_u, l_v))$

$SBPLT(l_u, l_v) = SBPLT(l_u, l_w) + SBPLT(l_w, l_v)$

3.3.2 Branch Delay Table (BDT)

Branch Delay Table contains the optimal delay of wire sized steiner tree connecting three nodes in N. If we know the buffer sizes and the locations of three nodes in N , we can find the optimal delay by $BDT(b_1, b_2, b_3, e, l_1, l_2)$ where b_1 , b_2 and b_3 are buffer sizes of the three buffers and e , l_1, l_2 are stem and branch lengths . These are shown in figure 3.4. We use dynamic programming approach to construct this table. This table will give the delay of a branch only when the degree t of the branch is two. For the cases of $t > 2$, we can use any fast steiner tree construction algorithm to get the solution.

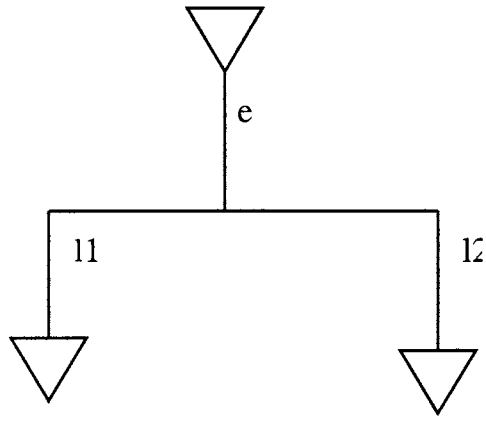

Figure 3.4 Branch

3.3.3 Stem Buffered Branch Construction

Assume that we know the positions and sizes of component drivers and receivers, delays of subtrees at the receivers. To make use of buffered path delay table and branch delay table/fast steiner tree construction algorithm, for constructing optimal stem buffered branch connecting driver to receivers, the position and size of branch driver should be

available. In our approach, for optimal branch driver position, we try all the locations present in the small grid space R which covers bounding box of the component driver and receivers of stem buffered branch. We place the branch driver at particular node in R and try all the buffer sizes available in buffer library. We size and position the branch driver such that, the maximum elmore delay at the receivers is minimized. For leaf buffer path, if we know the position and size of component driver, we can get the optimal routing using buffered path delay table.

3.4 Routing Tree Construction

In this section and the following section we explain how to size and position the component drivers and compute the delays of subtrees at each component driver. As we mentioned before, we can use any stochastic search algorithm to search for the routing tree. In our implementation, we use simulated annealing as search algorithm. We fix the positions of component drivers in simulated annealing. Assume that the positions of component drivers of all components present in routing tree are known. To fix the sizes of buffers, we use bottom-up dynamic program approach. We start from leaf buffer paths. For a leaf buffer path, we know the position of component driver. For the size, we try all the buffer sizes present in B for that component driver of leaf buffer path. As a result, we have B different routing solutions for that leaf buffer path. For stem buffered branch, solutions for the subtrees at each of the receivers must have been calculated already. So subtree at each receiver of stem buffered branch has B different solutions

corresponding to B different sizes for that receiver. Similar to leaf buffer path case, we try all the buffer sizes present in library for component driver, and get B solutions for stem buffered branch, each corresponding to B different sizes of that component driver. We can observe that at any component driver in routing tree, we have only B different solutions for the subtree driven by that component driver. Finally at the source, each receiver of the stem buffered branch driven by source have B different solutions for their respective subtrees. Hence we have B^{t_s} different solutions at the source , where t_s is the degree of stem buffered branch driven by source. Among these, final routing tree solution is the one which gives minimum of maximum elmore delay at each receiver.

3.5 Binary Tree Representation of Routing Tree

A general routing tree is a non binary tree. It is very difficult to represent and handle non-binary trees when compared to binary trees. Hence in our algorithm , we represent all the routing trees as binary trees. We transform a non binary tree into binary tree using by adding some **dummy nodes** to it. Final solution from our algorithm doesnot dependent on the binary tree representation of routing tree. The figure3.5 shows an example of this transformations. By adding dummy nodes, any non binary tree will have more than one binary tree representation. This introduces lot of redundancy into the solution space. We avoid this redundancy by putting a restriction that only the right child of any node can be dummy node. As we defined moves to make a real node to dummy node and a dummy node to a real node, the number of component buffers in

a routing tree is not fixed.We explain all the moves in next section.

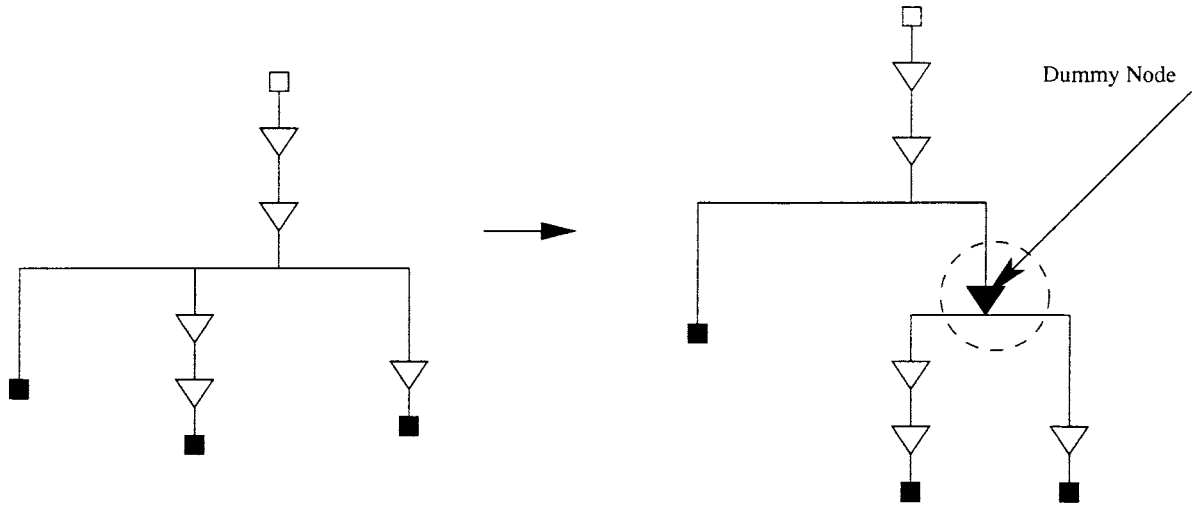

Figure 3.5 Transformation of Non binary tree to Binary Tree

3.6 Routing Tree Perturbation

In simulated annealing, we apply the following moves randomly.

1. **Move 1 - Component Driver Position change** :

   In simulated annealing, we change the positions of component drivers of stem buffered branch and leaf buffer path. For component driver of stem buffered branch, we randomly select one buffer node among 8 adjacent nodes and change the position of the component driver to that position. For component driver of leaf buffer path, we make a greedy move. We visit all the buffer nodes among 8 adjacent nodes of component driver and change its position to a node, where the sum of leaf buffer path delay and wire path delay from its parent is minimum.

2. **Move 2 - Swapping Of Sinks:**

   This is a topology changing move. In this move, we select component drivers of two leaf buffer paths (which can be sinks them selves when the length of leaf buffer path is zero) driven by two different parents and swap their parents.

3. **Move 3 - Rotation:**

   This is also a topology changing move. In this move, we have two types of rotations[5], one is left rotation and other is right rotation. The figure 3.6 shows these two operations. Using these operations we can get all binary trees that can be constructed with the given terminals. This confirms that we visit all the routing topologies in simulated annealing. When we are making right or left rotations, if a node is violating the restriction that, only right child of a parent node can be dummy, we change that dummy node to real node and then make the rotation. We explain how to make a dummy node to real node in the next move.

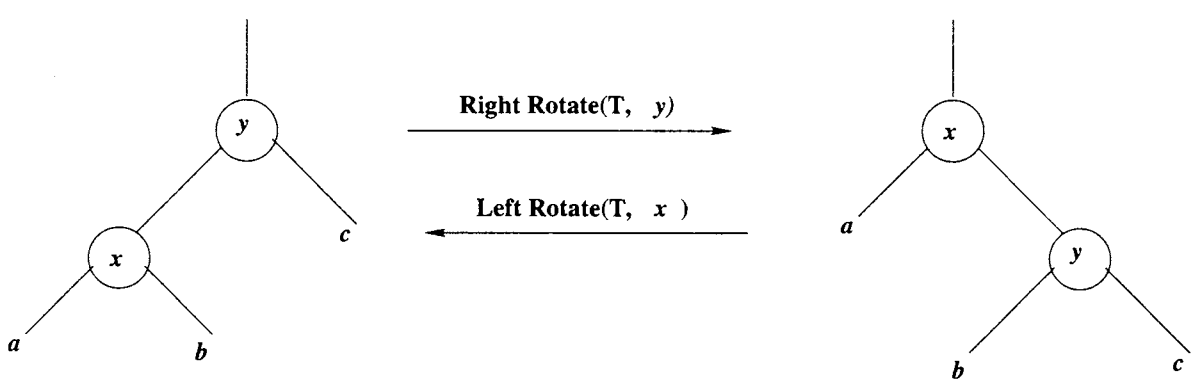

Figure 3.6 Rotation Operations

4. **Move 4 - Make Dummy and Make Real :**

   In this move, we select a buffer node randomly from set N and make that node dummy if it is a real node, make node real if it is dummy. When we make node dummy, we set a bit which indicates that the node is dummy. When we make a dummy node to real, we need to give that buffer a size and location. We give the size of buffer as minimum size that is available in B and for location we search the unblocked nodes near its left child (which cannot be a dummy node) and assign its location to that node.

CHAPTER 4. Complexity Analysis

For a given routing grid graph $G(V, E)$, runtime of our algorithm is calculated below,

4.1 Time Complexity for Lookup Tables

1. Shortest Wire Path Length Table and Shortest Buffered Path Length Table : We use Lee's algorithm to construct these tables. Runtime to construct these tables is $O(V^2)$.

2. Wire Path Delay Table : We use dynamic programming technique to do wire sizing. In our approach, we do wire sizing only if the path length between two nodes is less than critical length[14]. *critical length* is defined as the minimum wire length beyond which buffer insertion will help to reduce the interconnect delay. If path length exceeds the *critical length*, we connect those two nodes by thickest wire available in library W . Let L be the *critical length* between two nodes for which wire sizing is needed, then run time to construct this table is $O(B^2N^2)$.

3. Buffered Path Delay Table : The runtime for constructing this table with out pruning is $O(N^3B^3)$. Because of pruning technique that we are using to construct

this table, the runtime required is very less. Let T_b be the runtime required to build this table.

4. Branch Delay Table : We use dynamic programming technique to construct this table. Similar to Wire Path, when the branch length or stem length of a branch is greater than *critical length*, we use thickest wire present in the library to connect the driver to the receivers of branch. We are constructing this table only for branches with degree two. Runtime required to construct this table is given as $O(B^3L^3)$, L is the *critical length*. For branches with degree more than two, we can use any fast steiner tree construction algorithm. But for simplicity, we assume that the receivers are directly connected to the driver without any steiner point. We calculate delay for these branches by looking Wire Path Delay Table, with out any extra run time.

4.2 Complexity

Total run time of our algorithm is $O(V^2 + B^2N^2 + T_b + B^3L^3 + MC)$ where M is the number of iterations in simulated annealing and C is the time to compute delay of the routing tree in each iteration. We are doing buffer sizing in each iteration of Simulated Annealing, by using dynamic programming technique. Run time for each iteration of simulated annealing, C is $O(ST_l^2B^{t+1}R)$, where t is maximum degree of a branch in routing tree, S is the number of stem buffered branches present in routing tree, R is the grid space we search for branch driver location of each stem buffered branch, T_l is the

time to look up buffered path delay table or branch delay table. Usually T_l and S are small numbers and R is also small number when compared to N. From above expression it is clear that, our algorithm is very fast when compared to graph-RTBW. By neglecting non dominant terms we can express our runtime as $O(T_b + M(ST_l^2B^{t+1}R))$.

The memory space required for the look up tables is $O(V^2 + B^2N^2 + B^3L^3)$. The space required to store binary tree can be given by $O(B * (2k-1))$, where k is the number of sinks. Hence total memory required can be given by $O(V^2+B^2N^2+B^3L^3+B*(2k-1))$

.

4.3 Solution Space

The solution space of simulated annealing is given by $O(\frac{k!2^{2.543k}}{k^{1.5}}N^{(2k-1)}2^{(\frac{k-2}{2})})$, where k is the number of sinks. From[18] we know that tight bound on number of binary trees that we can construct is $\theta(\frac{k!2^{2.543k}}{k^{1.5}})$, where k is the number of leaves. As we have to search for $2k-1$ buffer locations, we have $N^{(2k-1)}$ options for different buffer locations. In our approach we make some nodes dummy to consider routing trees with fanout more than 2. When we are making a node dummy, we avoid the redundant trees by making a node dummy if and only if it is a right child. So we have $2^{(\frac{k-2}{2})}$ different topologies. Hence the total solution space can be given by $O(\frac{k!2^{2.543k}}{k^{1.5}}N^{(2k-1)}2^{(\frac{k-2}{2})})$.

CHAPTER 5. Experimental Results

We implemented FastRoute in C language and tested it on a Sun Ultra-2, 750MHz machine with 8GB memory. We tested graph-RTBW and SP-Tree also on the same machine.The driver resistance of the source is set to be 180Ω, and load capacitance of sink $23.4fF$ respectively. Among types of buffers used in the program, one has input capacitance of $23.4fF$ and output resistance of 180Ω. The intrinsic delay of buffers is set to be $36.4ps$. The resistance R_w and capacitance C_w for a wire with width x and length l are given by $R_w = r_w l/x$ and $C_w = c_a xl + c_f l$ respectively, where r_w is unit resistance, c_a is up-down wire capacitance per unit area. We use the parameters:$r_w = 0.076\Omega/\mu m$, $c_a = 0.024fF/\mu m$ and $c_f = 0.094fF/\mu m$. The output resistance R_b and input capacitance C_b for a buffer of size x are given by $R_b = r_b/x$ and $C_b = c_b x$ respectively, where r_b is unit buffer output resistance, c_b is unit buffer output capacitance. The intrinsic delay of buffer is same for all the buffer types. There are three choices of wires in our library:$\{x = 1, 2, 3\}$.Our grid is of size $17 \times 17mm^2$ with horizontal and vertical grid lines spaced at 0.5mm distance from each other.

The results reported are obtained by testing three programs on same specified machine. For graph-RTBW, we generated the trees and then calculated the elmore delay

for all the trees generated and reported these delay values to maintain the consistency with delay models of other two algorithms. All the testcases are randomly generated. All the test cases have buffer and wire blockages.

DATA	FastRoute						graph-RTBW[16]			
name	delay	CPU(s)			WL	buf	delay	CPU	WL	buf
	(ps)	LUT	SA-1	Total	(mm)		(ps)	(s)	(mm)	
NET4	1008	22.5	1.31	40.63	43	11	1008	170	44.5	13
NET5	937	22.5	3.04	52.94	45	11	937	305.56	47	14
NET6	1179	22.5	4.04	62.9	54.5	16	1157	663.06	72.5	22
NET8	1288	22.5	3.04	52.94	70	19	1281	4895	76	26
NET13	1149	22.5	6.06	83.16	77	20	*	$> 10hrs$	*	*
NET15	958	22.5	8.01	102.64	83	22	*	$> 10hrs$	*	*
NET18	1071	22.5	9.79	120.35	124	35	*	$> 10hrs$	*	*
NET21	1081	22.5	11.6	138.24	122	38	*	$> 10hrs$	*	*
NET23	1057	22.5	13.3	155.4	130	34	*	$> 10hrs$	*	*
NET25	1079	22.5	14.8	170	142	40	*	$> 10hrs$	*	*

Table 5.1 Comparison of FastRoute with graph-RTBW for $B = 1$ and $W = 1$

DATA	FastRoute						SP-TREE[9]			
name	delay	CPU(s)			WL	buf	delay	CPU	WL	buf
	(ps)	LUT	SA-1	Total	(mm)		(ps)	(s)	(mm)	
NET4	1008	22.5	1.31	40.63	43	11	1009	2	43	9
NET5	937	22.5	3.04	52.94	45	11	937	7.1	45	10
NET6	1179	22.5	4.04	62.9	54.5	16	1166	17	53.5	14
NET8	1288	22.5	3.04	52.94	70	19	1285	91	70.5	16
NET13	1149	22.5	6.06	83.16	77	20	1136	1387	79	18
NET15	958	22.5	8.01	102.64	83	22	941	7287	81.5	24
NET18	1071	22.5	9.79	120.35	124	35	1047.7	16729	120.5	33
NET21	1081	22.5	11.6	138.24	122	38	*	$> 10hrs$	*	*
NET23	1057	22.5	13.3	155.4	130	34	*	$> 10hrs$	*	*
NET25	1079	22.5	14.8	170	142	40	*	$> 10hrs$	*	*

Table 5.2 Comparison of FastRoute with SP-Tree for $B = 1$ and $W = 1$

Table 5.1 and Table 5.2 show the comparison between FastRoute , graph-RTBW and SP-Tree with single buffer and wire type. Here, we used $B = 1$ and $W = 1$, because

SP-Tree cannot handle wiresizing and graph-RTBW is implemented only for one buffer type . In our experiments, we run simulated annealing for 10 times and take the best result. All the testcases are randomly generated. As the grid for each testcase is same, the runtime for look up tabel construction is same for all the testcases. We can observe that, the time taken for simulated annealing is very less, because of the small solution space. Hence, once the grid is fixed we can construct look up tables once and use same set of look up tables to route all the nets present in the grid.For small testcases, we are much faster than graph-RTBW, but slower than SP-Tree. For, moderate size testcases, we are several hundred times faster than both the algorithms.Total CPU Time for our algorithm includes time for constructing lookup tables and running simulated annealing 10 times.As graph-RTBW minimizes only maximum delay, we have implemented our algorithm to optimize the maximum delay.But, in practice we can optimize minimum slack instead of maximum delay.

DATA	SP-Tree[9]				FastRoute					
name	delay (ps)	CPU (s)	wl (mm)	buf	delay (ps)	CPU(s) LUT	SA-1	TOTAL	wl (mm)	buf
NET4	848	9.3	44.5	12	851	145	3.924	184.24	43	13
NET5	787	33	45.5	12	790	145	8.025	225.25	45	13
NET6	977	71	52	17	987	145	14.8	293.06	52	20
NET8	1074	455	69	16	1091	145	24.37	388.71	72	24
NET13	954	5983	79.5	23	973	145	57.63	721.3	80	21
NET15	790	29000	85	28	804	145	66.37	808.72	87	20
NET18	*	$> 10hrs$	*	*	939	145	88.5	1030.45	129	37
NET21	*	$> 10hrs$	*	*	962	145	122.24	1367.4	144	36
NET23	*	$> 10hrs$	*	*	924	145	134.3	1487.77	138.5	37
NET25	*	$> 10hrs$	*	*	935	145	152.4	1668.5	152.5	43

Table 5.3 $B = 2, W = 1, Blockages = 11, grid = 17 \times 17mm^2$

In **Table 5.3** we compare our algorithm with SP-Tree for more than one buffer type. Even though we are slow for small test cases, we are very fast for moderate and large testcases. We can easily observe that our algorithm is scalable with problem size.

DATA	graph-RTBW[16]		FastRoute			
name	delay (ps)	CPU (s)	delay (ps)	CPU(s) LUT	SA-1	Total
NET4	918	171.23	918	22.5	1.91	41.6
NET5	851	310	851	22.5	2.61	48.6
NET6	1055	681	1061	22.5	3.314	55.64
NET8	1166	4971	1172	22.5	4.15	63.95
NET13	*	$> 10hrs$	1057	22.5	7.81	100.55
NET15	*	$> 10hrs$	885	22.5	8.9	111.51
NET18	*	$> 10hrs$	1061	22.5	11	132.35
NET21	*	$> 10hrs$	1049	22.5	13.6	158.2
NET23	*	$> 10hrs$	1092	22.5	15.91	181.59
NET25	*	$> 10hrs$	1041	22.5	17.9	201.49

Table 5.4 $B = 1, W = 3, Blockages = 11, grid = 17 \times 17mm^2$

In **Table 5.4** we compare our algorithm with graph-RTBW for more than one wire type. We handle wiresizing with out any increase in runtime. But, from Table 5.1, we can conclude that compared to graph-RTBW, we are always much better in resource consumption. When wiresizing is considered, our algorithm is several hundered times faster than graph-RTBW.

CHAPTER 6. Conclusions

In deep submicron designs, the interconnect delay is dominant factor in determining the circuit performance. In the last decade, a lot of research has been going on reducing interconnect delay and total interconnect length on a VLSI circuit. Buffer insertion and wire sizing have been shown to be effective techniques in reducing the interconnect delay. To get optimal interconnect, its very important to consider the wire blocks and buffer blocks simultaneously along with routing.

In this thesis we have presented a fast and efficient algorithm to construct a routing tree with simultaneous buffer insertion and wire sizing. While constructing the routing tree, we consider both buffer and wire obstacles present.The main idea of algorithm is to specify some important high-level features of the whole routing tree so that it can be broken down into several components. We apply stochastic search to find the best specification. Compared to topology search approaches and graph-RTBW, our algorithm is several hundreds times faster for moderate and large testcases. Our algorithm is scalable with problem size.We handle wire sizing with out any increase in runtime.

BIBLIOGRAPHY

[1] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H. Madden. "Performance optimization of VLSI interconnect layout". *INTEGRATION, the VLSI Journal*, 21:1–94, 1996.

[2] Takumi Okamoto and Jason Cong. "Buffered Steiner tree construction with wire sizing for interconnect layout optimization". In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, pages 44–49, 1996.

[3] C. Alpert, G. Gandham, M. Hrkic, J. Hu, A. Kahng, J. Lillis, B. Liu, S. Sapatnekar, A. Sullivan, and P. Villarubia. "Buffered Steiner Trees for difficult instances". in *Proc. Int. Symp. Physical Design.*, 2001, pp. 4-9.

[4] C. Alpert, C. Chu, G. Gandham, M. Hrkic, J. Hu, C. Kashyap, S. Quay, "Simultaneous driver sizing and buffer insertion using a delay penalty estimation technique", in *Proc. Int. Sym. Physical Design*, 2002, pp. 104-109.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", *Prentice - Hall,* 1997.

[6] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers", *Journal of Applied Physics*, 1948, 19:55-63.

[7] Rohini Gupta, Bogdan Tutuianu, Lawrence T.Pileggi, "The Elmore Delay as a Bound for RC Trees with Generalized Input Signals", *Proc. ACM/IEEE Design Automation Conference*, 1995, pp. 364-369.

[8] J.Hu, C. Alpert, S. T. Quay, and G. Gandham, "Buffer Insertion with Adaptive Blockage Avoidance", in *Proc. Int. Symp. Physical Design.*, 2002, pp. 92-97.

[9] M.Hrkic, J. Lillis, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages", in *Proc. Int. Symp. Physical Design.*, 2002, pp. 98-103.

[10] M.Hrkic, J. Lillis, "S-Tree: A technique for Buffered routing tree synthesis", *SASIMI*, 2001, pp. 242-249.

[11] M.Lai and D. Wong, "Maze routing with buffer insertion and wiresizing", in *Proc. ACM/IEEE Design Automation Conf.*, 2000, pp. 374-378.

[12] J . Lillis, C. K. Cheng, and T. T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model", in *Proc Int. Conf. Computer-Aided Design.*, 1996, pp. 134-139.

[13] J. Lillis, C. K. Cheng, T. T. Lin, C. Y. Ho, "Performance driven routing techniques

with explicit area/delay tradeoff and simultaneous wire sizing", in *Proc. ACM/IEEE Design Automation Conf.*, 1996, 395-400.

[14] R.Otten and R. Brayton, " Planning for performance" in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 122-127.

[15] J. Rubinstein, P. Penfield, and N. A. Horowitz, " Signal delay in RC tree networks", *IEEE Transactions on Computer-Aided Design.*, 1983, pp. 202-211.

[16] X. Tang, R. Tian, H. Xiang, and D. Wong, "A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints" in *Proc. Int. Conf. Computer-Aided Design.*, 2001, pp. 49-56.

[17] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, " Simultaneous routing and buffer insertion with restrictions on buffer locations", in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 96-99.

[18] Zion Cien Shen, C.N.Chu, "Bounds on Number of Slicing , Mosaic, and General Floorplans", To be appeared in *IEEE Transactions on Computer Aided Design of Integrated Circuits.*

[19] Semiconductor Industry Association, National Technology Roadmap for Semiconductors, 1997.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all who helped me in various aspects of my research and writing this thesis. First of all I want to thank my advisor Prof. Chris Chu for giving me an opportunity to work in his research group and explore the field of Electronic Design Automation. He always encouraged me and helped me whenever I needed. I would like to thank my other research group members for participating in useful discussions and for their invaluable suggestions.

I would also like to thank my friends Nitin, Bharath, Vineela, Rajesh, Nataraj, Sriram, Shantha and Lux for standing by me in good and bad times during my stay at Iowa State University.

Finally I would like to thank my family for their support and understanding.